Embedded.com

Li'l Bow Wow

By Jack G. Ganssle, Embedded Systems Programming
Jan 15, 2003 (2:12 PM)
URL: http://www.embedded.com/story/OEG20030115S0042

Although a watchdog timer is essential to reliability, a vulnerable dog does not a reliable system make. Here's how you can breed a champion.

Watchdog timers live on the boundary between hardware and software. A watchdog, along with other hardware interlocks found on some equipment, is an essential resource that protects users from rogue code, honest design errors, and acts of God. The humble watchdog timer (WDT) is the last resort when our embedded systems crash; it's the only chance we have to bring the system back online without manual user intervention.

But I believe our philosophy of watchdogging is quite flawed. Our misplaced faith in the perfection of our code leads too many to believe that crashes happen rarely if at all. As a result, the WDT design is usually an afterthought. My observations suggest that WDTs are called upon often over the life of a system-so their design had better be of the highest quality; higher even than the rest of the application, which may have crashed and invoked the WDT. If you expect your system to run reliably and can't count on a nearby user to hit the reset button when the code goes astray, you should add a killer watchdog.

I did a mini-survey of typical external and internal watchdogs last month, and all of them came up short. None met the exacting standards that I think are needed for high-reliability systems.

What constitutes an awesome watchdog timer? The perfect WDT should detect all erratic and insane software modes. It must not make any assumptions about the condition of the software or the hardware; in the real world, anything that can go wrong will. The ultimate WDT must bring the system back to normal operation no matter what went wrong, whether it was a software defect, a RAM glitch, or a bit flip from cosmic rays.

It's impossible to recover from a hardware failure that keeps the computer from running properly, but at the least the WDT must put the system into a safe state. Finally, it should leave breadcrumbs behind, generating debug information for the developers. After all, a watchdog timeout is the yin and yang of an embedded system. It saves the system, keeping customers happy, yet demonstrates an inherent design flaw that should be addressed. Without debug information, troubleshooting these infrequent and erratic events is close to impossible. What does all this mean in practice? Here's my take.

Best of breed

An effective watchdog is independent from the main system. Though all WDTs are a blend of interacting hardware and software, something external to the processor must always be poised like the sword of Damocles, ready to intervene as soon as a crash occurs. Pure software

Build a watchdog that monitors the entire system's operation. Don't assume that things are fine just because some loop or interrupt service routine runs often enough to tickle the WDT. The watchdog's software should look at a variety of parameters to ensure the product is healthy, kicking the dog only if everything is okay. What is a software crash, after all? Occasionally the system executes a HALT and stops, but more often the code vectors off to a random location, continuing to run instructions. Maybe only one task crashed. Perhaps only one is still alive-no doubt that which kicks the dog.

Think about what can go wrong in your system. Take corrective action when that's possible, but initiate a reset when it's not. For instance, can your system recover from exceptions like floating point overflow or divide by zero? If not, these conditions may well signal the early stages of a crash. Either handle these competently or initiate a WDT timeout. For the cost of a handful of lines of code, you may keep a 60 Minutes camera crew from appearing at your door.

It's a good idea to light an LED or otherwise indicate that the WDT kicked. A lot of devices automatically recover from timeouts; they quickly come back to life with the customer unaware a crash occurred. Unless you have a debug LED, how do you know if your precious creation is working properly or occasionally invisibly resetting? One outfit I consulted for complained that over time, and with several thousand units in the field, their product's response time to user inputs degraded noticeably. A bit of research showed that their system's watchdog properly drove the CPU's reset signal, and the code then recognized a warm boot, going directly to the application with no indication to the users that the time-out had occurred. We tracked the problem down to a floating input on the CPU that caused the software to crash-up to several thousand times per second. The processor was spending most of its time resetting, leading to apparently slow user response. A simple LED would have flagged the problem during debug, long before customers started yelling.

Everyone knows we should include a jumper to disable the WDT during debugging. But few folks think this through. The jumper should be inserted to enable debugging and removed for normal operation. Otherwise, if manufacturing forgets to install the jumper, or if it falls out during shipment, the WDT won't function. And there's no production test to check the watchdog's operation.

Design the logic so the jumper disconnects the WDT from the reset line (possibly though an inverter so an inserted jumper sets debug mode). Then the watchdog continues to function even while debugging the system. It won't reset the processor but will light the LED. (The light may come on when breakpointing and singlestepping, but should never come on during full-speed testing.)

In the doghouse

Most embedded processors that include high-integration peripherals have some sort of built-in WDT. Avoid using them except in the most cost-sensitive or benign systems. Internal WDTs offer minimal protection from rogue code. Runaway software may reprogram the WDT controller, many internal WDTs will not generate a proper reset, and any failure of the processor will make it impossible to put the peripherals into a safe state. A great WDT must be independent of the CPU it's trying to protect.

However, in systems that must use the internal versions, there's plenty we can do to make them more reliable. The conventional model of kicking a simple timer at erratic intervals is too easily spoofed by runaway code.

A pair of design rules leads to decent WDTs: prove the software is running properly by executing a series of unrelated things, all of which must work, before kicking the dog, and make sure that intervals, to further limit the chances that a runaway program deludes the WDT into avoiding a reset.

Perhaps it seems extreme to add an entire processor just for the sake of a decent watchdog. We'd be fools to add extra hardware to a highly cost-constrained product. Most of us, though, build lower volume, higher margin systems. A 50-cent part that prevents the loss of an expensive mission or that even saves the cost of one customer support call might make a lot of sense.

I'd hoped to conclude this discussion in just two columns, but there's more. Lots more. I'm passionate about reliable embedded systems, and watchdogs are an integral part of these. Stay tuned for next month's thoughts on watchdogs in multitasking systems and some ways to fix applications that only partially crash.

Jack G. Ganssle is a lecturer and consultant on embedded development issues. He conducts seminars on embedded systems and helps companies with their embedded challenges. Contact him at jack@ganssle.com.

Reference

For the basics on watchdog timers, see Murphy, Niall and Michael Barr, "Beginner's Corner: Watchdog Timers," *Embedded Systems Programming,* October 2001, p.79.